CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 7

1. (20 pts.) **Dijkstra's Algorithm.**

Solution:

(a) The order of vertices in which they are removed from the priority queue is s, a, c, d, b.

(b) As the order of vertices being removed from the priority queue needs to be same as before, we need to guarantee $distance(s,c) < distance(s,d) < distance(s,b)$. Changing $l(a,d)$ only potentially affects $distance(s,d)$ and is not possible to affect $distance(s,c)$ as well as $distance(s,b)$. So, we are certain that $distance(s,c) = 3$ and $distance(s,b) = 7$. Thus, our goal is to make sure $3 < distance(s,d) < 7$.

$$distance(s,d) = \begin{cases} l(s,a) + l(a,d) = 1 + l(a,d) & \text{if } 1 + l(a,d) \leq 6 \\ l(s,a) + l(a,c) + l(c,d) = 6 & \text{if } 6 \leq 1 + l(a,d) \end{cases}$$

With the consideration of $3 < distance(s,d) < 7$, the condition of the first case ($(s,a) \to (a,d)$ is the shortest path from s to d) is $3 < 1 + l(a,d) \leq 6$, and the the condition of the second case ($(s,a) \to (a,c) \to (c,d)$ is the shortest path from s to d) is still $6 \leq 1 + l(a,d)$. Overall, as we consider both cases, the range of $l(a,d)$ is $(2, \infty)$.

2. (20 pts.) **Unique Shortest Path.**

Solution: This can be done by slightly modifying Dijkstra's algorithm. The array `usp[·]` is initialized to `true` in the initialization loop. The main loop is modified as follows, with the central change being an extra equality check between dist(v) and dist(u)+$l(u,v)$.

```
while H is not empty do
  u = GetMin()
  Remove(H, 0)
  for all (u,v) ∈ E do
    if dist(v) = dist(u) + l(u,v) then
      usp[v] = false
    end if
    if dist(v) > dist(u) + l(u,v) then
      dist(v) = dist(u) + l(u,v)
      usp[v] = usp[u]
      DecreaseKey(H, position[v], dist[v])
    end if
  end for
end while
```

This algorithm works because, for any node u that is removed from the heap, no new shortest paths to u can be discovered. This is due to the edge weights all being positive. Then, for every edge (u,v), if going through u represents a new shortest path to v, we know that the shortest path to v can be unique only if the shortest path to u is also unique. Finally, if a path to v is uncovered that has equal length to the existing shortest path to v, we know there are at least two shortest paths to v. This could be reset by later discovering a shorter path to v in the future.

3. (20 pts.) **Start Node Negative Edges.**

Solution: Dijkstra's algorithm would work.

Proof: Consider the proof of Dijkstra's algorithm. The proof depended on the fact that if we know the shortest paths for a subset $S \subseteq V$ of vertices, and if (u,v) is an edge going out of S such that v has the minimum estimate of distance from s among the vertices in $V \backslash S$, then the shortest path to v consists of the (known) path to u and the edge (u,v). We can argue that this still holds even if the edges going out of the vertex s are allowed to be negative. Let (u,v) be the edge out of S as described above. For the sake of contradiction, assume that the path claimed above is not the shortest path to v. Then there must be some other path from s to v which is shorter. Since $s \in S$ and $v \notin S$, there must be some edge (i,j) in this path such that $i \in S$ and $j \notin S$. But then, the distance from s to j along this path must be greater than the estimate of v, since v had the minimum estimate. Also, the edges on the path between j and v must all have non-negative weights since the only negative edges are the ones out of s. Hence, the distance along this path from s to v must be greater than the estimate of v, which leads to a contradiction.

4. (20 pts.) **Shortest Path in Currency Trading.**

Solution:

a) Represent the currencies as the vertex set V of a complete directed graph G. To find the most advantageous ways to converts c_s into c_t, you need to find the path $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$ maximizing the product $r_{i_1,i_2} r_{i_2,i_3} \cdot \cdots \cdot r_{i_{k-1},i_k}$. This is equivalent to minimizing the sum $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}})$. Hence, it is sufficient to find a shortest path in the graph G with weights $w_{ij} = -\log r_{ij}$. Because these weights can be negative, we apply the Bellman-Ford algorithm for shortest paths to the graph , taking s as origin.

b) Just iterate the updating procedure once more after $|E||V|$ rounds. If any distance is updated, a negative cycle is guaranteed to exist, i.e. a cycle with $\sum_{j=1}^{k-1}(-\log r_{i_j,i_{j+1}}) < 0$, which implies $\prod_{j=1}^{k-1} r_{i_j,i_{j+1}} > 1$, as required.

5. (20 pts.) **Faster All-Pairs Shortest Path.**

(a) The total weight of a path is the sum of the weights of its edges. When we expand the reweighting formula, the intermediate h values cancel each other out. For example, edge (v_2,v_3) has $-h(v_3)$, but the next edge (v_3,v_4) has $+h(v_3)$. The only h values missing a pair to cancel with are $+h(v_0)$ and $-h(v_k)$:

$$\sum_{i=0}^{k-1} \hat{w}(v_i,v_{i+1}) = \sum_{i=0}^{k-1} (w(v_i,v_{i+1}) + h(v_i) - h(v_{i+1})) = h(v_0) - h(v_k) + \sum_{i=0}^{k-1} w(v_i,v_{i+1})$$

We are left with the sum of the original weights, which is the total weight of the original path, plus $h(v_0) - h(v_k)$. Note that any path from v_0 to v_k shares those endpoints, so $h(v_0) - h(v_k)$ is constant and the total weights of every path maintain their relative ordering. Therefore, the original shortest path is still the shortest path after reweighting.

(b) This part is similar to (a). The total weight of a cycle is the sum of the weights of its edges. The only difference is that here, the starting and ending points are equal, so $h(v_0) = h(v_k)$. This means that every h value will cancel.

$$\sum_{i=0}^{k-1} \hat{w}(v_i,v_{i+1}) = \sum_{i=0}^{k-1} (w(v_i,v_{i+1}) + h(v_i) - h(v_{i+1})) = \sum_{i=0}^{k-1} w(v_i,v_{i+1})$$

Reweighting preserves the exact total weight of every cycle. Therefore, negative reweighted cycles can occur if and only if the original cycle was also negative.

(c) The central observation is that because h is defined using shortest paths from s, the shortest path from s to v must at most equal in length to the shortest path from s to u plus the distance between u and v. This is true because either the path through u is the shortest path to v, in which case they are equal, or the path through u is not the shortest path to v, in which case the shortest path to v must be even shorter. This inequality can be formalized as $h(v) \leq h(u) + w(u,v)$. Subtracting $h(v)$ gives $w(u,v) + h(u) - h(v) \geq 0$. This is exactly our formula for $\hat{w}(u,v)$, so $\hat{w}(u,v) \geq 0$.

(d) We are comparing Floyd-Warshall's $O(|V|^3)$ to this algorithm's $O(|V|^2 \log |V| + |V||E|)$. $|V|^2 \log |V|$ is always preferable to $|V|^3$; the problem is $|V||E|$. In a graph with $O(|V|^2)$ edges (such as a complete graph), $O(|V||E|) = O(|V|^3)$. So, the condition for this algorithm to be faster is that the number of edges must not be $\Omega(|V|^2)$. For example, in a tree, there are $O(|V|)$ edges and Johnson's algorithm is dominated by $O(|V|^2 \log |V|)$, which is much better than $O(|V|^3)$. In short, we say that Johnson's Algorithm is preferable for sparse graphs.

6. (10 pts.) **Extra Credit.**

Solution: We will modify Dijkstra's algorithm to solve this problem. Dijkstra keeps a set A, and $dist(u)$ for all $u \notin A$. A contains nodes whose shortest path from s are already computed. For $u \notin A$, $dist(u)$ stores the shortest $s \to u$ path among the paths that start at s, and use only nodes in A before ending at u. At each iteration, Dijkstra adds the u with the smallest $dist(u)$ to A, and update the $dist(v)$ for all v where $(u,v) \in E$.

In particular, we will change the order of how the nodes are added to A. We first recognize the SCC of our graph treating roads as bi-directional edges. If two nodes are connected by roads, then they must be in the same SCC, and the additional constraint about planes ensures that no plane edge will be inside a SCC. Thus the edges across different SCCs are exactly the plane routes.

We first find the topological order of the metagraph of SCCs, and for each node u in an SCC, let $r(u)$ be the index of the SCC in the topological ordering. It is easy to see that if $r(u) = r(v)$, then any path from u to v must be all roads, and if $r(u) > r(v)$, there won't be any path from u to v.

The modification we make to Dijkstra is that on each iteration, we include the u with the smallest $r(u)$, and among the nodes with the same $r(u)$, we include the one with the smallest $dist(u)$.

Proof of correctness: We will argue when we add u to A, $dist(u)$ is the shortest path distance from s to u. Consider any $s \to u$ path we haven't considered (i.e. a path visits some node not in A before ending at u, let v be the first such node on the path). Since the algorithm adds u instead of v in this iteration, we must have either $r(v) > r(u)$ or $r(v) = r(u), dist(v) \geq dist(u)$. In the first case, there is no path from v to u (contradiction!), and in the second case, the part of the path from v to u must be all roads (thus positive), and $dist(v) \geq dist(u)$ guarantees the initial part of the path from s to v is already no shorter than the paths we have considered for $dist(u)$, thus the entire path won't be shorter. Either way, it is safe to say we have already computed the shortest path for u. Once we established this property, the correctness of the algorithm follows from the proof of Dijkstra.

Remark: Note that although Dijkstra's algorithm can not be used generally for finding shortest paths on a graph with negative edges, there are some special graph structure with negative edges (like this problem or problem 1 of this homework) that Dijkstra's algorithm still works.

Running time: it is the same as Dijkstra's running time, since the preprocessing (i.e. finding SCCs, lin-

earization) is linear time, and the rest is the same complexity as Dijkstra.

Rubric:

Problem 1, 20 pts

(a) 8 points: 2 points for each correctly ordered vertex (except s as it's specified as the starting vertex).

(b) 12 points
4 points: correctly identify that the goal is $distance(s,c) < distance(s,d) < distance(s,b)$
2 points: correctly calculate $distance(s,c)$ and $distance(s,b)$
4 points: correctly recognize the two possible cases for $distance(s,d)$
2 points: perform correct calculations for the inequalities with respect to these two cases and obtain the correct final result

Problem 2, 20 pts

- 5 pts for saying Dijkstra's Algorithm works
- 15 pts for a correct proof (contradiction or otherwise showing that the normal correctness of Dijkstra's Algorithm is unchanged.

Problem 3, 20 pts

- 5 pts Modified Dijkstra's Algorithm
- 15 pts
  if dist(v) = dist(u) + $l(u,v)$ **then**
  usp[v] = `false`
  end if

Problem 4, 20 pts

(a) 12 pts:
- 6 pts for reducing the problem to a shortest path problem.
- 6 pts for solving the problem with the Bellman-Ford algorithm, and showing why it works

(b) 8 pts:
- 4 pts for relating the presence of anomaly to a negative cycle in the graph
- 4 pts for showing how we can find negative cycle efficiently ($O(|V||E|)$, such as with the Bellman-Ford Algorithm)

Problem 5, 20 pts

(a) 6 pts for showing all internal h values cancel, leaving only $h(v_0)$ and $h(v_k)$, which are constant for all possible paths.

(b) 4 pts for showing all h values cancel in a cycle, preserving the original weight.

(c) 6 pts for using the triangle inequality to show the reweight formula cannot be negative.

(d) 4 pts for explaining the given algorithm works better for small $|E|$

Problem 6, 10 pts

(a) 5 pts for mentioning the SCCs of the graph and saying that all road edges lie entirely in SCCs, and plane edges only connect two different SCCs (plane edges do not lie in a SCC)

(b) 4 pts for correctly modifying the Dijkstra algorithm, and designing the efficient algorithm.

(c) 1 pts for showing the correctness of efficient algorithm.